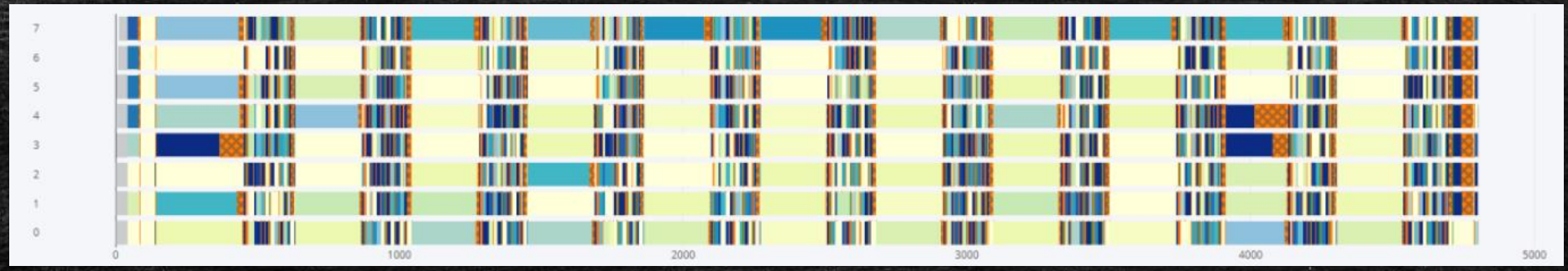
# Root causing MPI workloads imbalance issues via scalable MPI Critical Path analysis

Artem Shatalin, Vitaly Slobodskoy and Maksim Fatin
Huawei, Computing Application Acceleration Technology Center

# My HPC workload is running too slow…

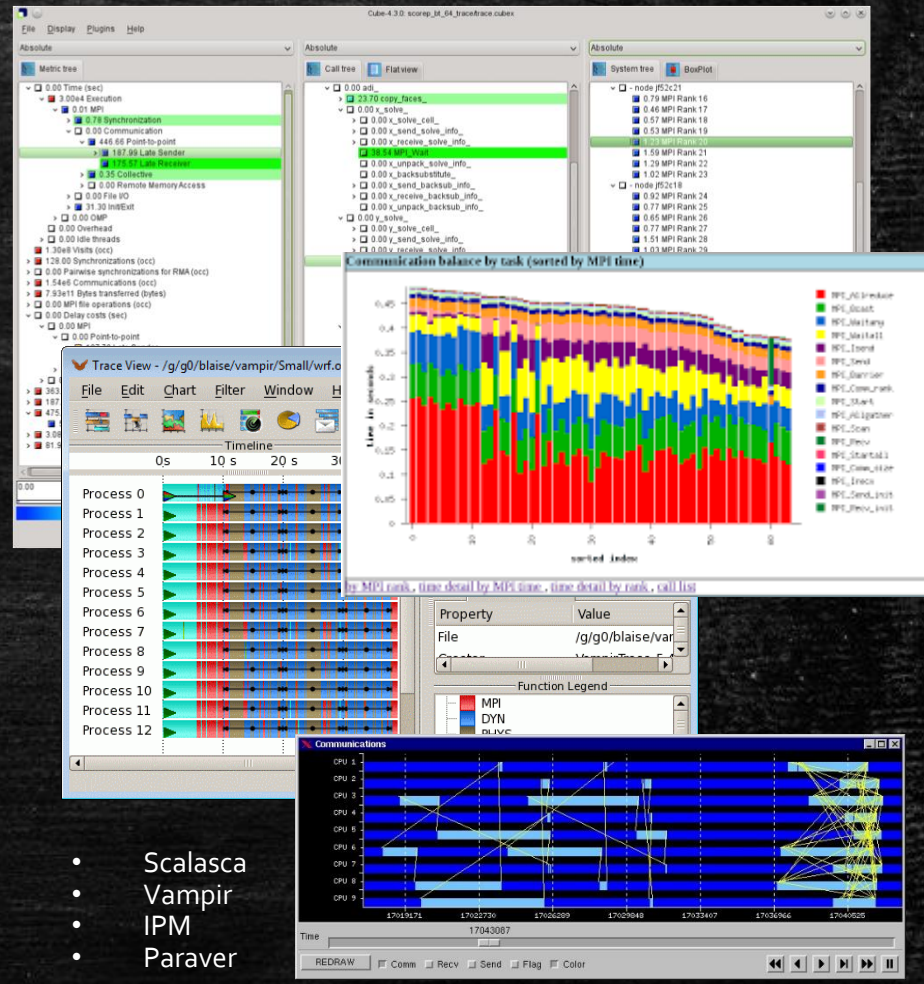
- Workflow imbalance is one of the most typical issues limiting HPC app performance and scaling




- How to measure imbalance? … on a big HPC cluster?
- How to minimize it?

# Existing methods

- MPI Tracing
  - Can capture overall time spent within MPI functions
  - Overtime visualization of time spent in computing/MPI runtime
  - No direct information from the MPI runtime about wait(imbalance)/transfer distinguishing
- Classical hotspot-based performance analysis
  - Often misleading for MPI applications because hotspots optimization might not actually cause any speedup, but just increase the time ranks spent on waiting for each other




- Scalasca
- Vampir
- IPM
- Paraver

# Hardware Sampling based Hotspots

All the data captured within a single PMI is a *sample* (*s*), $s \in S$ – the set of all the samples captured during performance analysis data collection with set of counters $C=\{c_1, c_2 \ldots, c_p\}$, $|C|=p$.

***Aggregation*** is an accumulation of sample values:

$$t = \{v_j | j \in 1..p\}, v_j = \sum_{s \in S, c(s)=c_j, j \in 1..p} value(s) \quad (1)$$

***Grouping*** is one or more sample attributes:

$$G = (A^1, A^2, \ldots, A^l), A^j \in A, j \in 1..l, 1 \le l \le k \quad (2)$$

***Grouping value*** for sample $g(s, G) = (a_1(s), a_2(s), \ldots, a_l(s))$

All the possible combinations of grouping values for grouping G and set of samples S

$$U(S, G) = \left\{ u = (a_1, a_2, \ldots, a_l) \middle| u \in \prod_{j=1}^{l} A^j, \exists\, s \in S: a_j(s) = a_j, j \in 1..l \right\} \quad (3)$$

***Aggregation of samples*** *S* by grouping *G* is $T = \{(u, v_1, v_2, \ldots, v_p)\}$ (4)

***Metric:*** $M_\theta = F_\theta\left(v_{i_1}, v_{i_2}, \ldots, v_{i_{q(\theta)}}\right), 1 \le i_j \le p, j \in 1..q(\theta)$

Table *H*(*S*,*G*,*M*) with rows representing items of aggregation of samples sorted by the value $m_o$ of primary metric $M_o$ in descending order is called ***hotspots.***

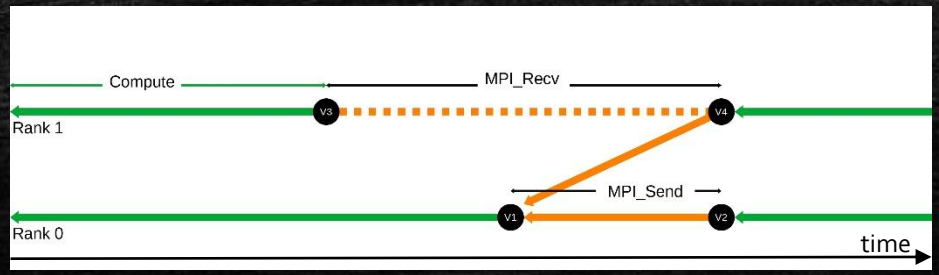
Sample characteristics:
**timestamp** *time(s)*,
**counter identifier** $c(s) \in C$,
**counter value** *value(s)*
**set of attribute values** $a_i(s) \in A_i, A_i \in A, |A| = k, i \in 1..k$.

| Function | Module | CPU Time(s) |
|---|---|---|
| hmca_bcol_basesmuma_bcast_k... | hmca_bcol_basesmuma.so | 2838.6540 |
| uct_dc_mlx5_iface_progress_ll | libuct_ib.so.0.0.0 | 2795.7913 |
| uct_mm_iface_progress | libuct.so.0.0.0 | 2327.4292 |
| opal_progress | libopen-pal.so.40.30.1 | 2090.3369 |

**Hotspots example**

# Program Activity Graph*

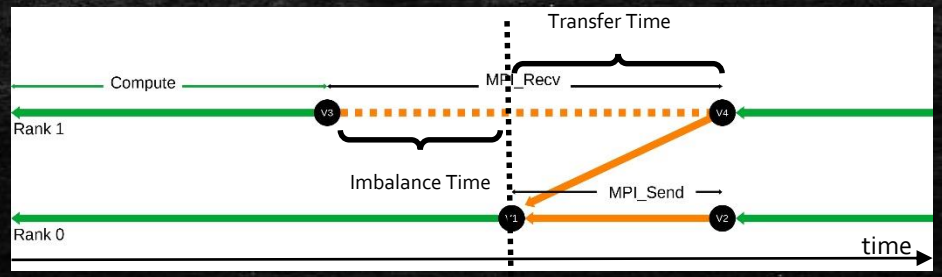
- Program Activity (PA) - non-overlapping individual job which has duration
  - There is precedence relationship among the PAs – some PAs must be finished before others can start
- Program Activity Graph is a directed, weighted, acyclic graph
  - Vertices represent beginnings and endings of PAs associated with particular communication events (e.g. send/receive) in a program
  - **Green edges** represent the duration of PA
  - **Yellow edges** are communication edges connecting endings and beginnings of PAs with precedence relationship (the shortest is marked solid, others are dotted)
  - The direction of PAG is from right to left (from the ending of app execution to begin)




* C.-Q. Yang, B. P. Miller. Critical Path Analysis for the Execution of Parallel and Distributed grams. 8th Int'l Conf. on Distributed Computing Systems. June 1988, San Jose, Calif., pp. 366-375.

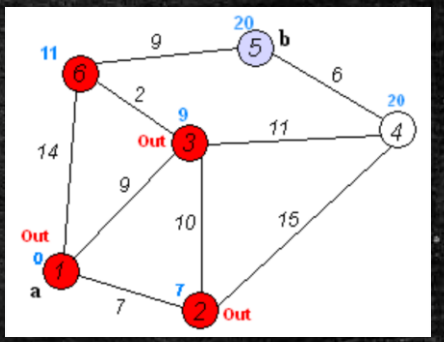
# Critical Path Analysis

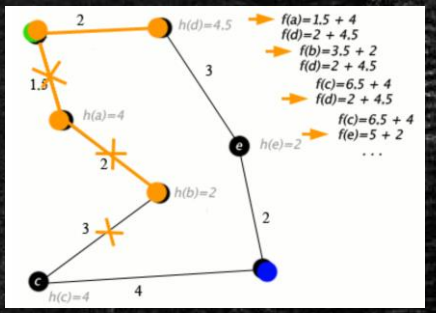
- **The Critical Path** is the longest path in the graph
  - As for PAG with communication edges Critical Path is the longest path with the longest compute time (shortest communication time)
- There is no Imbalance Time (e.g. spinning) on the Critical Path
  - **Imbalance Time** is the time rank spent on waiting other ranks when it finished computing part earlier
- Shortening the time of Critical Path leads to the application Elapsed Time reduction
  - Performance tool should be able to provide information helping user to optimize performance of activities on the Critical Path



# Known ways to find the Critical Path in DAG

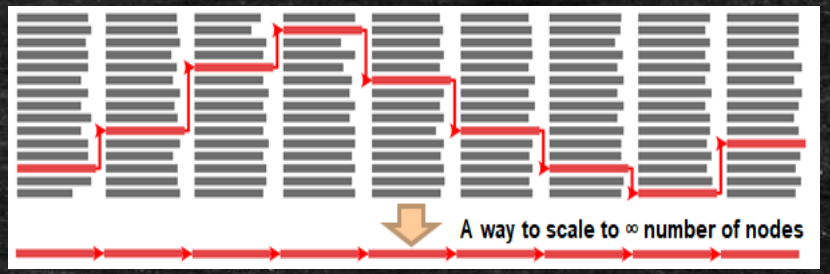
- Shortest path algorithms with negative weights:
  - Dijkstra's algorithm, O(number of edges), **sequential**
  - A* algorithm, efficiently operates under particular assumptions, **sequential**
  - Delta-stepping and its modifications, **parallel**
- Critical Path analysis methodologies for the parallel programs:
  - [*C.-Q. Yang and B. P. Miller., 1988*] with parallel version of longest path algorithm based on [*K. M. Chandy and J. Misra., 1982*] – the first attempt to apply CP for analysis of parallel programs
  - An approach for automatic search of optimization suggestions using Critical Path analysis of task graph built on top of GASPI [*D. Grunewald and C. Simmendinger, 2013*] applications was considered in [*C. Herold et al., 2017*].
  - One of the most recent papers [*D. D. Nguyen and K. L. Karavanic, 2021*] underlines the importance of Critical Path for the efficient performance analysis of the real-world HPC workloads. A new metric called "Workflow Critical Path" is defined to characterize distributed workloads.



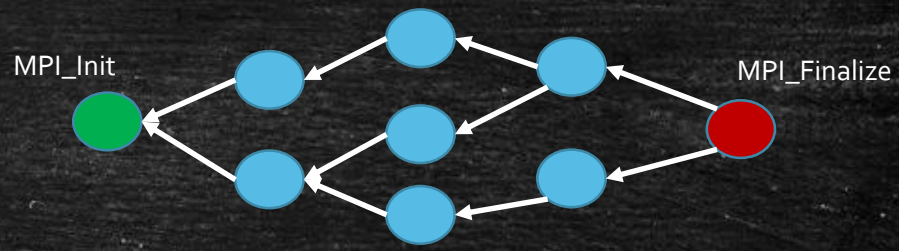

- https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
- https://en.wikipedia.org/wiki/A*_search_algorithm

# Proposed solution

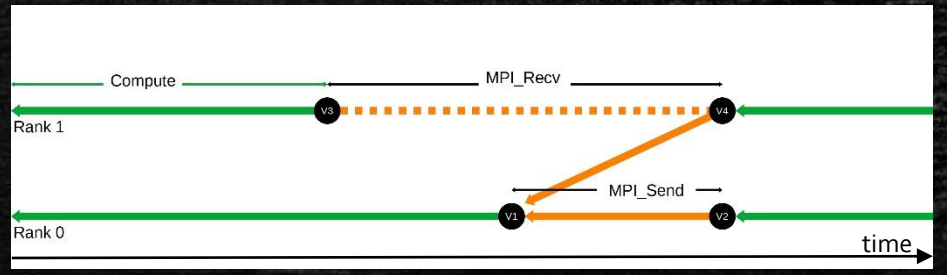



- Combine critical path data with hardware sampling data in order to:
  1. Represent the most critical information for the further optimization (any reduction of Critical Path will lead to application elapsed time reduction).
  2. Limit the amount of performance data analyzed to the size equal to performance data size collected from a single MPI rank only due to Critical Path relates to just a single rank at any moment of application execution time. This naturally supports any scale since the amount of performance data to be analyzed is always limited and depends on the application execution time only.
- We propose a highly scalable MPI calls replay based solution for constructing the Critical Path with **less than 5%** of collection overhead and **less than 5%** of application elapsed time spent on post-processing independently on the number of ranks.
- Our approach has been tested on various real-world workloads and stays within performance targets even on relatively high scale (available to us for testing).

# PAG construction rules




1. Graph vertex is a beginning or ending of MPI call on a particular rank.
2. Graph vertex gets weight, which equals to the elapsed time from the beginning of the application execution to the beginning/ending of corresponding MPI call.
3. Every vertex has only one outgoing edge. All the edges are directed from the node having bigger weight to the one having smaller weight. Edge representing communication time goes to the vertex with the highest weight (rank with the latest further MPI call start time).
4. Everything executed outside of MPI call is treated as a useful compute job (Program Activity).
5. Program Activity edges stay on the same rank.



# Critical Path finding algorithm

- Runtime (Data Collection):
  1. Trace all the relevant MPI calls with arguments capture for enabling further replay
- Post-processing (done within MPI_Finalize call of the application):
  2. Reconstruction of MPI communicators used by application.
  3. Exchange timings information between P2P senders and receivers via MPI P2P calls replay (from sender to receiver) and further collective-based exchange (from receiver to sender).
  4. Graph edges creation, replay of MPI collective calls.
  5. Finding Critical Path: retrieve time intervals of the path within the corresponding ranks via traversing Program Activity Graph from the rank with the latest MPI_Finalize call time.

**Lemma 1**: There is only one path on the last step of the algorithm.

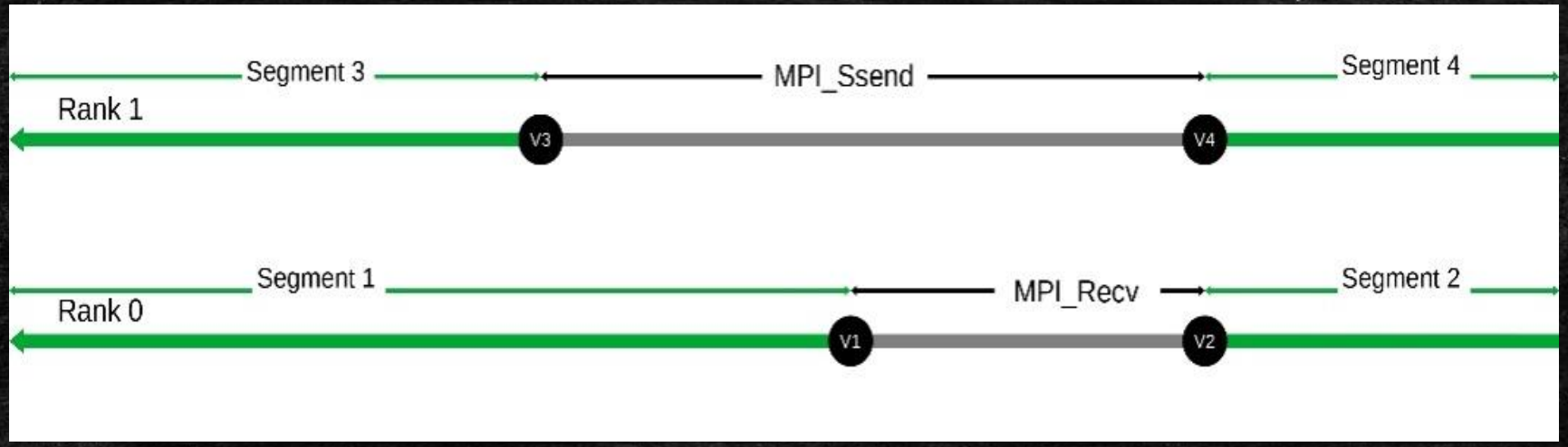
**Proof**: As every node has only one outgoing edge and graph is acyclic (graph construction rule 3), there is only one path in the graph from every node representing MPI_Finalize. Since only a single start node is chosen based on MPI_Finalize start time, there is only one path found.

# 1. Data Collection

- PAG building requires tracing of all the MPI calls within every rank of the target application
  - Use PMPI and LD_PRELOAD mechanisms for tracing all the relevant MPI calls and capture required arguments. Target app recompilation is not required.
  - MPI calls related to communicators management are traced in order to reconstruct communicators (step 2) for further replay of MPI calls on the post-processing stage
  - Synchronous and asynchronous MPI calls related to communication between ranks are captured with:
    - all the required arguments for further replay
    - timing information (start/end time using MPI_Wtime).

# 3. Exchange timings information between P2P senders and receivers

- Every P2P call has to retrieve a timing information about its pair call. This happens in 2 stages:
  1. Replay of P2P calls and Wait/Waitall calls related to asynchronous P2P calls. Receivers get information about senders on this stage.
  2. Use MPI collective calls in order to distribute information about receivers to every sender.

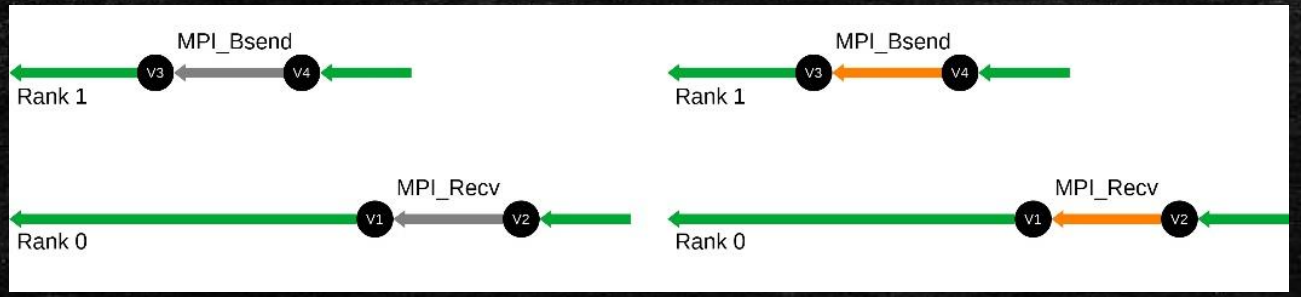



Typical MPI send/receive pattern

# 4. Graph edges creation

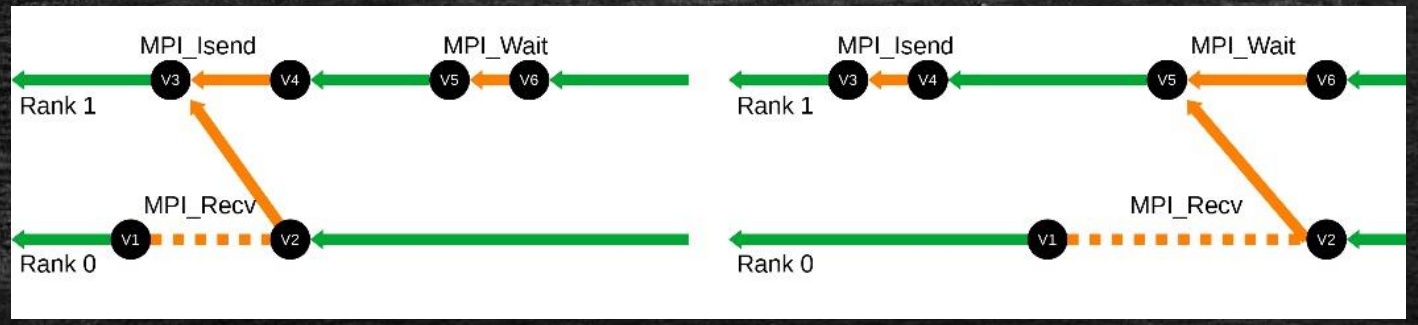
- The primary edge creation rule is based on the PAG construction rule 3: consider $k$ time intervals of related MPI calls $[b_i; e_i], i \in 1..k$, for every $e_i$ create an edge $(e_i \rightarrow b)$ where $\exists j: b = b_j, e_i > b_j, \nexists l: b_l > b_j, j \in 1..k, l \in 1..k$.


Creating graph edges for P2P calls

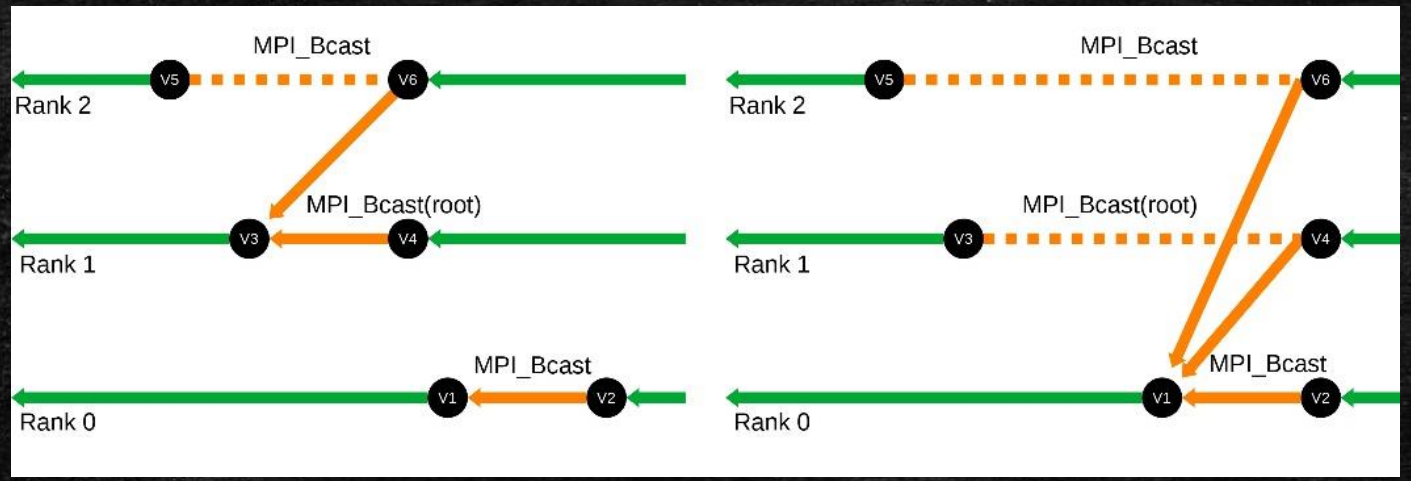



Caching is involved within P2P calls

# 4. Graph edges creation



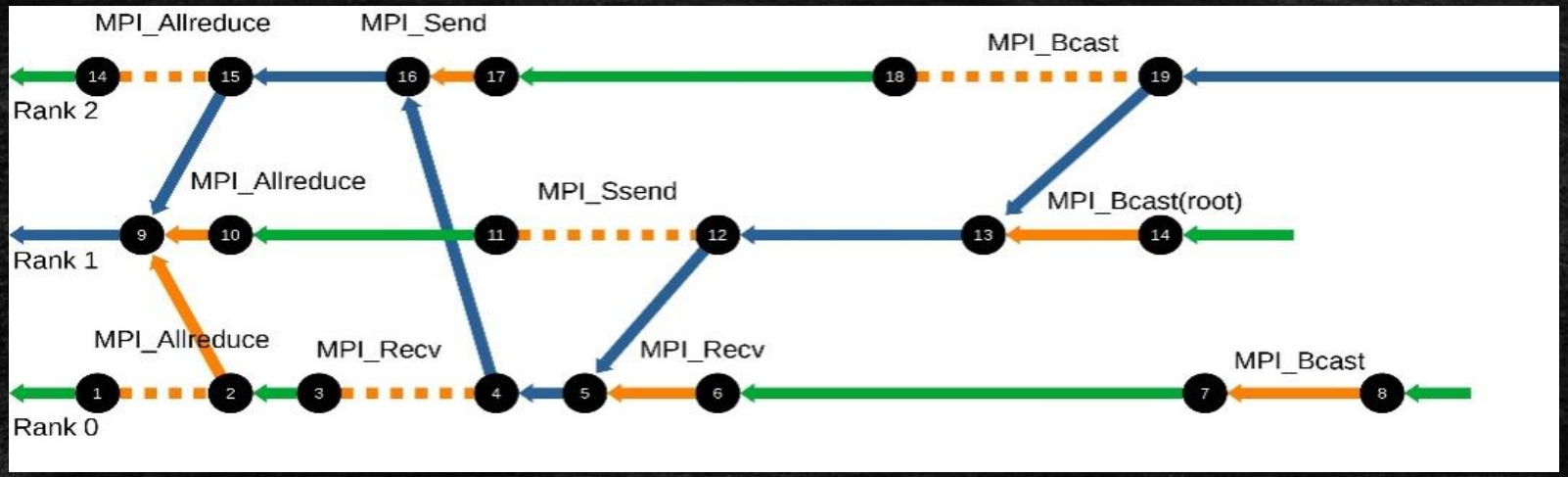

Asynchronous P2P calls




Synchronous collective calls

# 5. Finding the Critical Path

- In order to find the Critical Path we need to traverse through edges of the whole PAG.
  - According to Lemma 1, there is only one path from the latest node on every rank. So, the starting traversal point is the latest MPI_Finalize call across all the ranks. Traversing is done using P2P calls where ranks not on critical path are waiting on message receive, the one on critical path is sending message when critical path migrates to another rank and waits for further message receive:

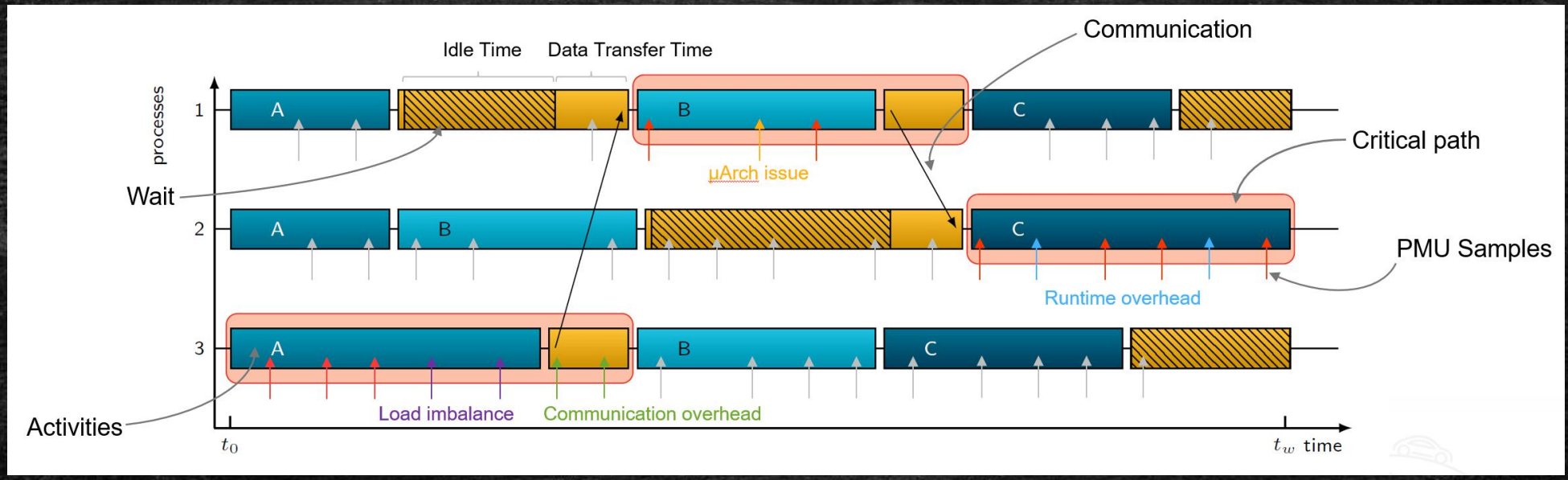



Critical Path example

# Algorithm features

- Efficiently utilizes all the available computing resources naturally inheriting the cluster topology knowledge from the application.
- Every rank keeps and maintains information about its own graph nodes and edges only.
- Post-processing elapsed time depends on the application elapsed time and is supposed to be **less than 5%** of elapsed time of the application due to the replay-based nature of the algorithm – MPI calls from the application are replayed only without computing portion of the application, the amount of data transferred within replay has small fixed amount for every MPI call. The post-processing overhead depends on the frequency of MPI calls done by the application.

# PMU Samples aggregation on Critical Path




- Hotspots on the Critical Path naturally highlight activities having the most significant influence on the application elapsed time. Optimization of the hotspots on Critical Path obviously leads to the reduction of application elapsed time.

# Hotspots on Critical Path

Quantum Espresso benchmark

**Top Hotspots**

| Function | Module | CPU Time(s) | Inst Retired | CPI |
|---|---|---|---|---|
| hmca_bcol_basesmuma_bcast_k... | hmca_bcol_basesmuma.so | 13383.5000 | 93319394951143 | 0.3729 |
| ucp_worker_progress | libucp.so.0.0.0 | 12456.9659 | 69298900225456 | 0.4674 |
| hmca_bcol_basesmuma_barrier... | hmca_bcol_basesmuma.so | 10216.3521 | 79180406179086 | 0.3355 |
| uct_mm_iface_progress | libuct.so.0.0.0 | 4788.1269 | 28096224988430 | 0.4431 |
| zgemm_ | pw.x | 4519.1347 | 12829183732610 | 0.9159 |

**Top Hotspots on MPI Critical Path**

| Function | Module | CPU Time(s) | Inst Retired | CPI |
|---|---|---|---|---|
| zgemm_ | pw.x | 15.1443 | 42899112983 | 0.9179 |
| ztrsm_ | pw.x | 3.2306 | 7691043444 | 1.0921 |
| dlaebz_ | pw.x | 2.8353 | 2478888508 | 2.9738 |
| zher2k_ | pw.x | 2.7822 | 8033314443 | 0.9005 |
| zgemv_ | pw.x | 2.1234 | 5357958074 | 1.0304 |

NAS Parallel benchmark (NPB) BT

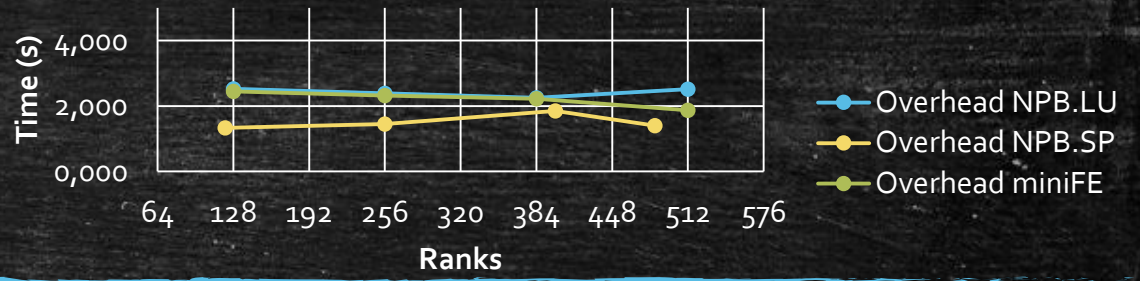
**Top Hotspots**

| Function | Module | CPU Time(s) | Inst Retired | CPI |
|---|---|---|---|---|
| uct_dc_mlx5_iface_progress_ll | libuct_ib.so.0.0.0 | 295.3117 | 1400019640511 | 0.5484 |
| binvcrhs_ | bt.C.x | 182.3859 | 862089470040 | 0.5501 |
| compute_rhs_ | bt.C.x | 164.1930 | 372248709623 | 1.1468 |
| x_solve_cell_ | bt.C.x | 137.2237 | 454555473911 | 0.7849 |
| y_solve_cell_ | bt.C.x | 136.8946 | 516508431287 | 0.6891 |

**Top Hotspots on MPI Critical Path**

| Function | Module | CPU Time(s) | Inst Retired | CPI |
|---|---|---|---|---|
| compute_rhs_ | bt.C.x | 0.2853 | 801341496 | 0.9257 |
| binvcrhs_ | bt.C.x | 0.2664 | 1646724591 | 0.4206 |
| x_solve_cell_ | bt.C.x | 0.2565 | 951072004 | 0.7013 |
| z_solve_cell_ | bt.C.x | 0.2408 | 900313343 | 0.6953 |
| y_solve_cell_ | bt.C.x | 0.2329 | 942058909 | 0.6428 |

# Performance Evaluation

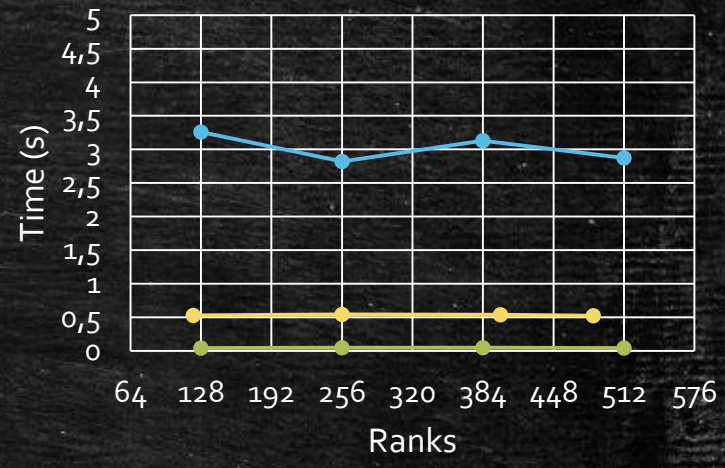



NPB BT

| Nodes | Ranks | Problem size | Elapsed time (s) | Elapsed time under collector(s) | Overhead | | Postprocesing | | MPI calls | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | s | % | s | % | Total | Per rank |
| 1 | 121 | 224x224x224 | 137.611 | 138.945 | 1.334 | 0.97 | 0.529 | 0.38 | 8,005,118 | 66,158 |
| 2 | 256 | 320x320x320 | 130.621 | 132.071 | 1.450 | 1.11 | 0.545 | 0.42 | 24,631,808 | 96,218 |
| 4 | 400 | 384x384x384 | 132.891 | 134.741 | 1.850 | 1.39 | 0.537 | 0.40 | 48,103,628 | 120,259 |
| 4 | 484 | 408x408x408 | 114.041 | 115.439 | 1.398 | 1.23 | 0.522 | 0.46 | 64,028,360 | 132,290 |

NPB LU

| Nodes | Ranks | Problem size | Elapsed time (s) | Elapsed time under collector (s) | Overhead | | Postprocessing | | MPI calls | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | s | % | s | % | Total | Per rank |
| 1 | 128 | 256x256x256 | 67.834 | 70.361 | 2.527 | 3.73 | 3.261 | 4.81 | 71,235,432 | 556,527 |
| 2 | 256 | 320x320x320 | 61.425 | 63.811 | 2.386 | 3.88 | 2.823 | 4.60 | 184,370,232 | 720,196 |
| 3 | 384 | 384x384x384 | 70.071 | 72.321 | 2.250 | 3.21 | 3.129 | 4.47 | 335,724,808 | 874,283 |
| 4 | 512 | 408x408x408 | 68.320 | 70.837 | 2.517 | 3.68 | 2.872 | 4.20 | 478,295,000 | 934,170 |




miniFE

| Nodes | Ranks | Problem size | Elapsed time (s) | Elapsed time under collector (s) | Overhead | | Postprocessing | | MPI calls | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | s | % | s | % | Total | Per rank |
| 1 | 128 | 1024x512x512 | 120.074 | 122.522 | 2.448 | 2.04 | 0.044 | 0.04 | 899,008 | 7,024 |
| 2 | 256 | 1024x1024x512 | 122.572 | 124.886 | 2.314 | 1.89 | 0.046 | 0.04 | 1,977,536 | 7,725 |
| 3 | 384 | 1024x1024x768 | 121.861 | 124.084 | 2.223 | 1.82 | 0.048 | 0.04 | 2,912,448 | 7,585 |
| 4 | 512 | 1024x1024x1024 | 124.244 | 126.113 | 1.869 | 1.50 | 0.043 | 0.04 | 4,350,016 | 8,496 |

WRF

| Nodes | Ranks | Elapsed time (s) | Elapsed time under collector (s) | Overhead | | Postprocessing | | MPI calls | | Overhead per call (us) |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | | s | % | s | % | Total | Per rank | |
| 8 | 256 | 5992.799 | 6037.222 | 44.423 | 0.74 | 6.831 | 0.11 | 386,975,194 | 1,511,622 | 29.388 |
| 16 | 512 | 3032.759 | 3075.891 | 43.132 | 1.42 | 7.236 | 0.24 | 812,547,288 | 1,587,006 | 27.178 |
| 32 | 1024 | 2079.178 | 2125.290 | 46.112 | 2.22 | 7.254 | 0.35 | 1,859,041,028 | 1,815,470 | 25.399 |
| 48 | 1536 | 1210.573 | 1247.913 | 37.340 | 3.08 | 7.102 | 0.59 | 2,684,833,958 | 1,747,939 | 21.362 |
| 64 | 2048 | 1091.817 | 1127.626 | 35.809 | 3.28 | 7.447 | 0.68 | 3,833,675,393 | 1,871,912 | 19.129 |

# vs Scalasca

| Benchmark | | NPB LU | | miniFE |
|---|---|---|---|---|
| Problem size | | 384x384x384 | 320x320x320 | 1024x512x512 |
| Scale (nodes x ranks) | | 1x128 | 1x128 | 1x128 |
| Elapsed time (s) | | 195.24 | 121.85 | 120.08 |
| Under collection | Our tool | 199.95 **(+2.41%)** | 122.64 **(+0.64%)** | 125.52 **(+2.04%)** |
| | Scalasca | 255.38 **(+30.80%)** | 147.25 **(+21.41%)** | **Failure** |
| Post- processing (s) | Our tool | 3.29 **(1.69%)** | 2.89 **(2.37%)** | 0.044 **(0.04%)** |
| | Scalasca | **Out of memory** | 86.45 **(60.52%)** | **Failure** |
| Collected data size | Our tool | 4.1GB | 3.3GB | 57MB |
| | Scalasca | **96GB** | **64 GB** | **Failure** |

# Conclusion

- ✓ We have created a novel scalable and robust approach for root causing MPI Imbalance issues in the MPI applications in order to improve efficiency of MPI parallel applications performance analysis.
  - ✓ New algorithm for building Program Activity Graph and finding the Critical Path in the Program Activity Graph of the MPI application has been developed, which scales well and doesn't require any complex operations on data collection.
  - ✓ A novel approach of PMU data aggregation to the hotspots on Critical Path has been proposed.
  - ✓ It supports ideal analysis scalability due to the limitation of analyzed PMU data to the amount of data collected from a single rank.
  - ✓ The runtime and post-processing cost of the analysis is negligible which has been confirmed by experiments involving real-world parallel workloads.
  - ✓ The runtime overhead of the proposed approach stays within just 5% related to wrapping of all the relevant MPI calls and capturing required data for further replay.
  - ✓ Critical Path analysis overhead also stays within just 5% of application elapsed time and doesn't depend on the number of ranks.
  - ✓ Overall, Critical Path analysis stays within 5% of runtime overhead which is much less than any existing solution working on the real-world applications.

# Backup

# Performance Evaluation

- The results were collected on the **clusters with 4 and 96 Huawei Taishan 2280v2 compute nodes**:
  - 2x64 core Hisilicon Kunpeng 920 CPU, 2.6GHz
  - 256GBDRAM, DDR4
  - Storage: PCI-e NVME SSDs
  - Interconnect: NVIDIA® (Mellanox) Infiniband Connect X6 100Gb adapters, 100Gb switch
  - Operating system: CentOS 8
  - Compiler: gcc/gfortran 11.2
  - MPI: OpenMPI 4.1.4
- **Benchmarks used for tool performance characteristics evaluation:**
  - **NAS Parallel Benchmark BT (NPB BT)**: solver for synthetic system of nonli-near partial different equations using block tridiagonal algorithm
  - **NAS Parallel Benchmark LU (NPB LU)**: solver for synthetic system of nonli-near partial different equations using Lower-Upper symmetric Gauss-Seidel algorithm
  - **MiniFE**: sparse linear system solver using a simple un-preconditioned conju-gate-gradient algorithm
  - **WRF**: Weather Research and Forecasting Model – a numerical weather prediction and atmospheric simulation system